**Test i Qualitat del Software**

**MASTERMIND**

<Adrián Moreno Gimeno - 1365146>

<Daniel Muñoz Vidal - 1332367>

**Functionality**: < Function which check if the return of a clue is correct.>

**Location**: <Clue.java, Clue & getClue>

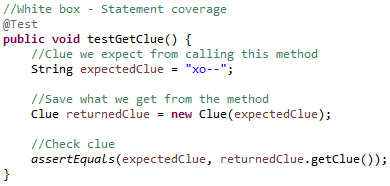
**Test**: <ClueTest.java, ClueTest & testGetClue>

< In this test we want to check that the correct value is returned by the function getClue.

It is a very short test but it is useful to make sure that the rest of the functions will not fail because of this.

We tested the method with white box test:

*Statement coverage*: with the method“coverage as JUnit test” we have checked that the execution goes over all lines in the code.>



**Functionality**: <Function which create a clue for each code the player enters>

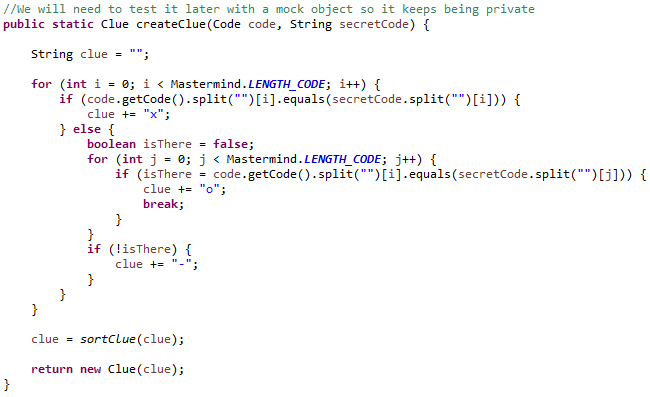
**Location**: <Clue.java, Clue & createClue>

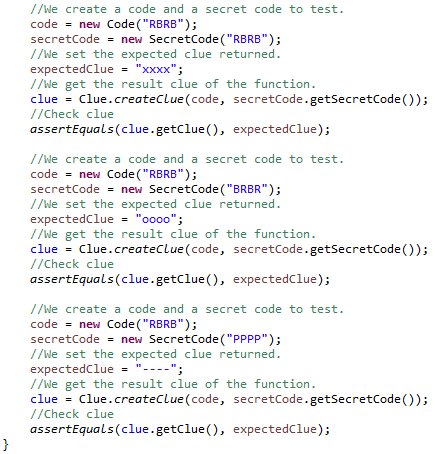
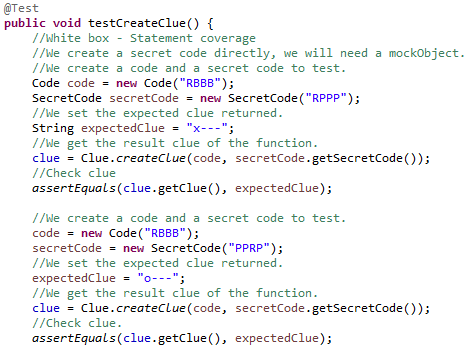
**Test**: <ClueTest.java, ClueTest & testCreateClue>

<We expect to check if the clue has been created correctly with this test. So we give the expected clue directly and we check if the clue that has been generated is the same than the clue we entered as expected.

We tested the method with white box test:

*Statement coverage*: with the method“coverage as JUnit test” we have checked that the execution goes over all lines in the code>





**Functionality**: <In this method, a certain clue is sorted >

**Location**: <Clue.java, Clue & sortClue>

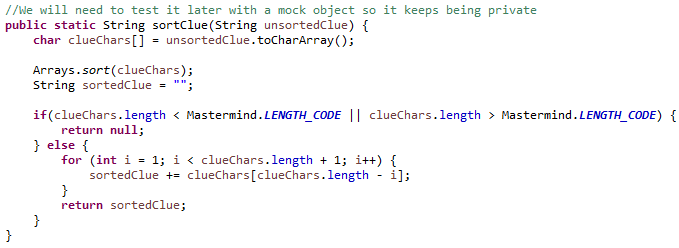
**Test**: <ClueTest.java, ClueTest & sortClue>

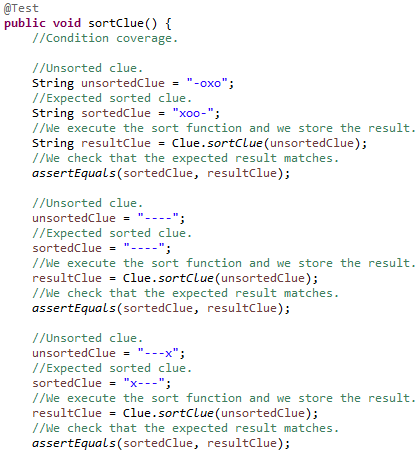
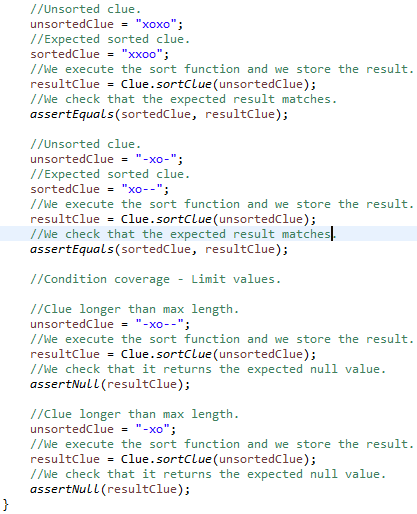
< In this test what you want to check is that given a clue, it is ordered correctly. In order to test it, we have created several clues to test, each different: -oxo, ----, --- x, xoxo, -xo-. In this way we check that in all cases the result obtained and the expected one are equal.

We tested the method with white box and black box tests:

*Statement coverage*: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code.

*Condition coverage*: having as limit values ​​a clue with length = 5 and another with length = 3 we can check that indeed, the method returns a null to us because the input values ​​are incorrect. It therefore goes through all the conditions of the code.>



**Functionality**: < Function which check the return of a code is correct.>

**Location**: <Code.java, Code & getCode>

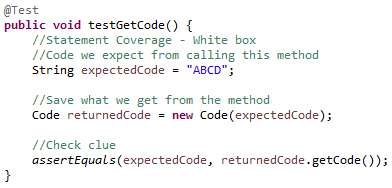
**Test**: <CodeTest.java, CodeTest & testGetCode>

<In this test we want to check that the correct value is returned by the function getCode.

It is a very short test but it is useful to make sure that the rest of the functions will not fail because of this.

We tested the method with white box test:

*Statement coverage*: with the method“coverage as JUnit test” we have checked that the execution goes over all lines in the code.>



**Functionality**: <In this method, the code is added to the GameBoard and is verified>

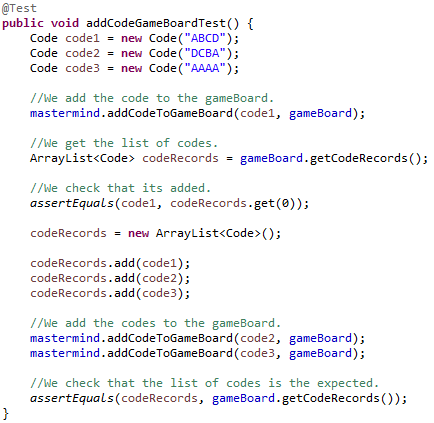
**Location**: <Mastermind.java, Mastermind & addCodeToGameBoard>

**Test**: < MastermindTest.java, MastermindTest & addCodeGameBoardTest >

<With this test we simply check that different codes are correctly added to the game board and ensure that this is not a problem for future tests.

We tested the method with white box tests:

*Statement coverage*: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code.>



**Functionality**: < In this method, the clue is added to the GameBoard and is verified>

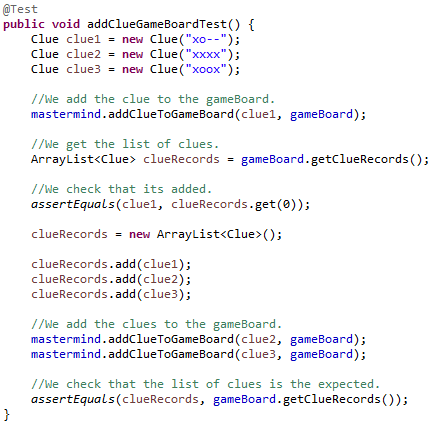
**Location**: <Mastermind.java, Mastermind & addClueToGameBoard>

**Test**: < MastermindTest.java, MastermindTest & addClueGameBoardTest >

<With this test we simply check that different clues are correctly added to the game board and ensure that this is not a problem for future tests.

We tested the method with white box tests:

*Statement coverage*: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code.>



**Functionality**: <Method that is responsible for increasing the number of attempts in the game.>

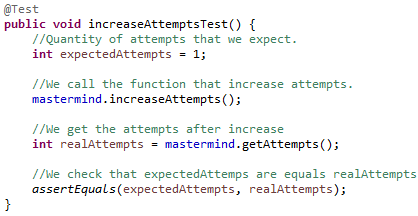
**Location**: <Mastermind.java, Mastermind & increaseAttempts>

**Test**: <TestMastermind.java, TestMastermind & increaseAttemptsTest>

<In this test what you want check is that given a certain number of attempts, this is correctly increased by one. In order to test this, given an attempt number ‘0’ and an expected attempt number ‘1’, we increase the number of attempts and check that the attempt has increased.

We tested the method with white box tests:

*Statement* *coverage*: with the “coverage as JUnit test” method we checked that the execution goes through all possible lines of code>



**Functionality**: <Method that is responsible for checking the number of attempts in the game>

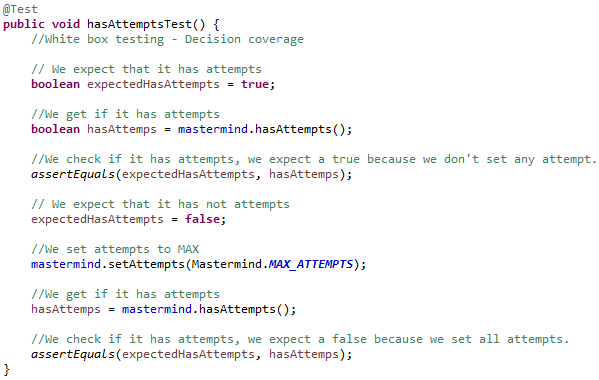
**Location**: <Mastermind.java, Mastermind & hasAttempts>

**Test**: <MastermindTest.java, MastermindTest & hasAttemptsTest>

<In this test, what you want to check is that given an initial number of attempts, it is checked whether or not the maximum number of attempts has been reached. In order to test this, given an attempt number N we check if this attempt number is within or outside the range of allowed attempts.

We tested the method with black box test:

*Decision coverage*: with the *“coverage as JUnit test”* method we have verified that the execution goes through all the possible conditions>



**Functionality**: <Method that is responsible for entering the code and clue and check the state of the game>

**Location**: <Mastermind.java, Mastermind & enterCode>

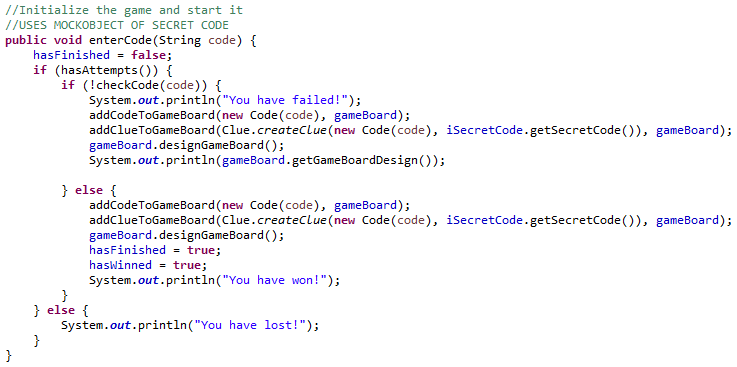
**Test**: <MastermindTest.java, MastermindTest & enterCodeTest>

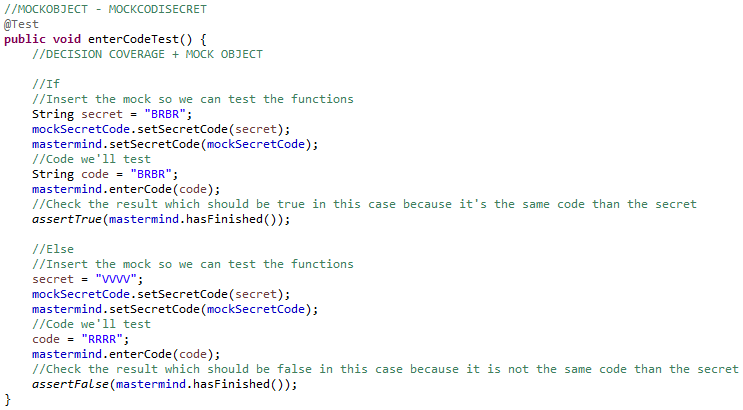
<In this function, the codes given by the player are entered and it is checked if that code is equal or not to the code given previously. They are also added to the history and a hint is generated so that the player has new information about the code he has entered.

We tested the method with black box tests:

*Decision* *coverage*: with the *“coverage as JUnit test”* method we have verified that the execution goes through all the possible conditions of the code

*Mock* *Object*: We create a mock object of secret code, this way a secret code is not created randomly and we can perform the relevant tests.*>*





**Functionality**: <Method that lets the player select the difficult of the game and changes it.>

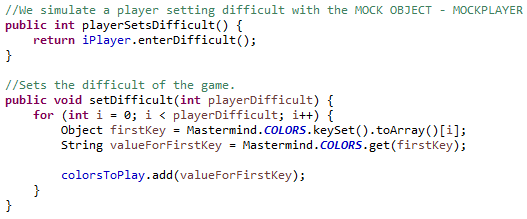
**Location**: <Mastermind.java, Mastermind & playerSetsDifficult>

**Test**: <MastermindTest.java, MastermindTest & playerSetsDifficultTest>

*<* Function that serves to test that the player can select a difficulty and depending on the level of difficulty are added more or less colors in addition to having more or less combinations. We also do some loop testing trying different levels of difficulty. The picture below is just a resume, the function itself has more tests not only the two that appear in the picture.

We tested the method with white box test:

*Mock* *Object*: We create a mock object of secret code, this way a secret code is not created randomly and we can perform the relevant tests. On the other hand, we also create a mock object of the player and in this way we can imitate the interaction by console*>*



**Functionality**: <Method that is responsible for letting the player new codes and check the result>

**Location**: <Mastermind.java, Mastermind & playerPlaysGame>

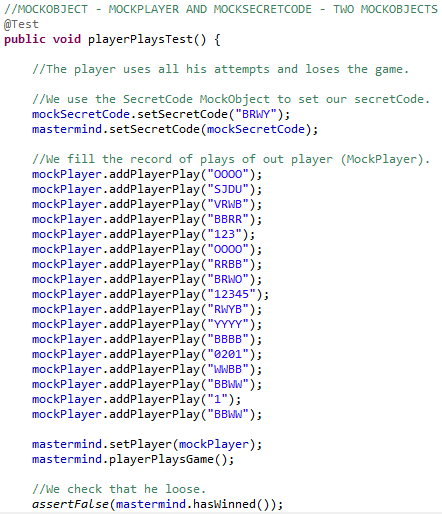
**Test**: <MastermindTest.java, MastermindTest & playerPlaysGameTest>

*<*In this test, what we do is simulate three games (in the pictures below we only added two because of the space, but in the code there are the three of them) : one where the player enters codes until he loses, another where he enters codes until he wins on the last attempt and another where he wins halfway through the game. Once these input codes have been given, check if the result is as expected.

We tested the method with white box tests:

*Decision* *coverage*: with the "coverage as JUnit test" method we have verified that the execution goes through all the conditions of the code.

*Mock* *Object*: We create a mock object of secret code, this way a secret code is not created randomly and we can perform the relevant tests. On the other hand, we also create a mock object of the player and in this way, we can imitate the interaction by console*>*

**Functionality**: <Method that does the path covered test>

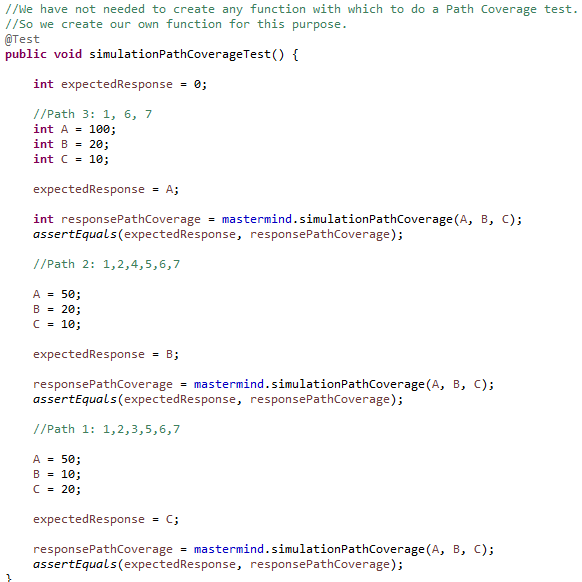
**Location**: <Mastermind.java, Mastermind & playerPlaysGame>

**Test**: <MastermindTest.java, MastermindTest & playerPlaysGameTest>

*<*The function is just a simulation of the path covered test, it’s the only purpose of this test.

We tested the method with white box test:

*Path* *coverage*: we have designed out test case such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of what's called a control flow graph of an application*>*



**Functionality**: <This method checks the validity in terms of content, length… of a code>

**Location**: <Player.java, Player, checkEnteredCode>

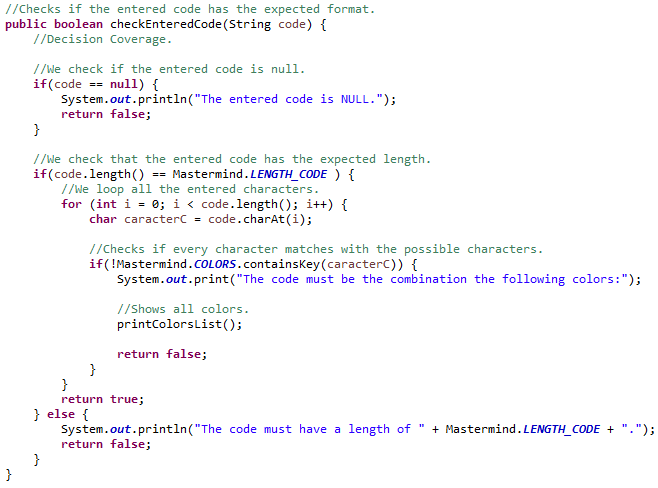
**Test**: <PlayerTest.java, PlayerTest and checkEnteredCodeTest.>

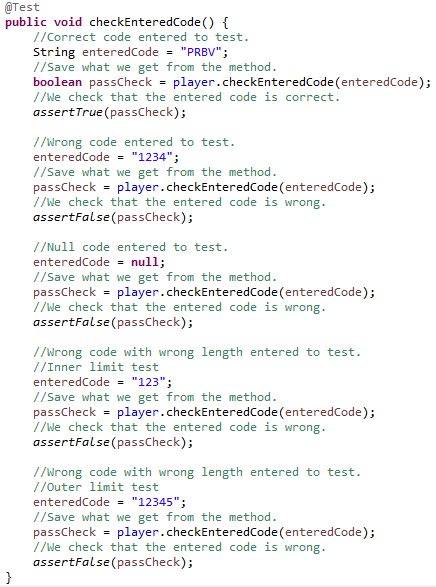
<In this test what you want to check is that given a code, it meets all the corresponding conditions. In order to test it, we have created several codes to test: a correct one, one that does not correspond to the possible colors (numerical code), a shorter than normal (less than 4), a longer one (more than 4) and a of null. In this way we can check that the test is actually fulfilled in all possibilities.

We tested the method with white box tests:

*Statement* *coverage*: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code.

*Decision* *coverage*: we have verified that the boundary values ​​(3 and 5), and the inner value (4), which are lengths of the code, do not give errors therefore it goes through all the conditions>





**Functionality**: <This method checks if the entered difficult has the expected format>

**Location**: <Player.java, Player, checkDifficult>

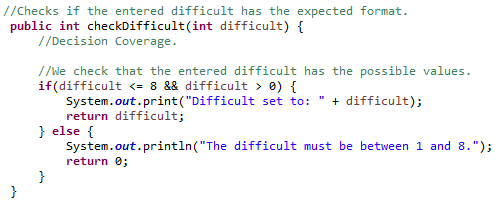
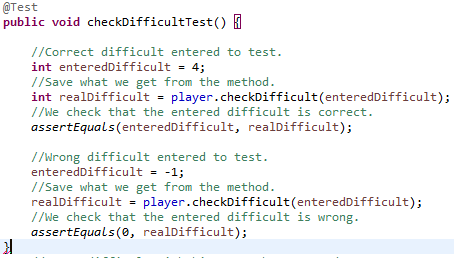
**Test**: <PlayerTest.java, PlayerTest and checkDifficultTest>

<In this test what you want to check is that the difficult entered has the format that was expected. In order to test it, we have created several difficulties to test: some of them are correct (between 1-8) and some of them are wrong (-1). The test has a lot of difficulties but we have only added a few in the picture.

We tested the method with white box tests:

*Statement* *coverage*: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code.

*Decision* *coverage*: we have verified that the boundary values ​​(3 and 5), and the inner value (4), which are lengths of the code, do not give errors therefore it goes through all the conditions>

**Functionality**: <Utility function to check if a string can be converted to integer>

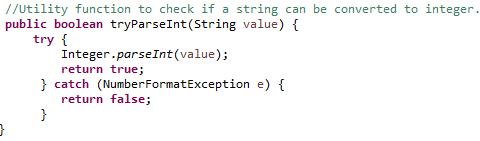
**Location**: <Player.java, Player, tryParseInt>

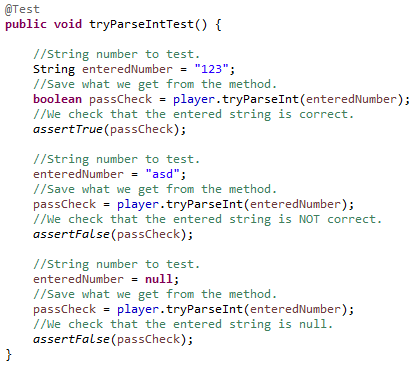
**Test**: <PlayerTest.java, PlayerTest and tryParseIntTest>

<This is just a utility function so we can check if a string can be converted to integer or not. The test is really quick and there’s no need to explain more about it.

We tested the method with white box tests:

*Statement* *coverage*: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code. >





**Functionality**: <In this method it is checked if a certain code given by the player matches the winning secret code.>

**Location**: <SecretCode.java, SecretCode, checkSecretCode>

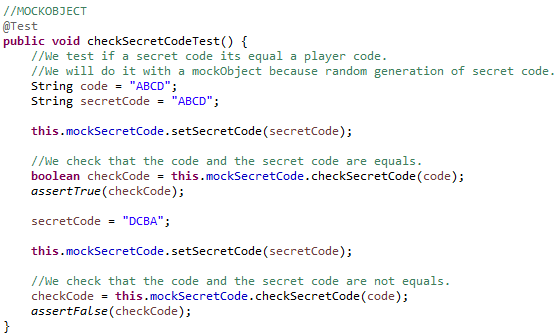
**Test**: <SecretCodeTest.java, SecretCodeTest and checkSecretCodeTest.>

<In this test what we want to check is that given a code and a code, the function returns to us if these two codes are equal.

We tested the method with white box tests:

*Statement* *coverage*: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code.

*Mock* *Object*: We create a secret code mock object, this way no secret code is created randomly and we can perform the relevant tests.>



Mock Classes:

We have created two mock classes because we need them for the tests. This way we ensure that we don’t change anything in our base code and it still work to pass all the tests. We have also created different interfaces for these mocks and the corresponding classes because we need them for the tests.

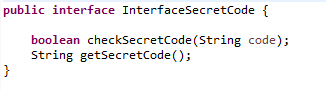
We’ll add below all the mock classes and will mention the corresponding tests that affect each mock class.

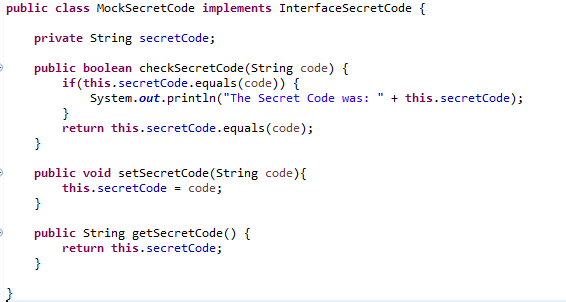
* **MockSecretCode**:

We have created a MockObject of the CodiSecret because the original class creates a competently random Code and we cannot test it.

As we do not want to modify this class and we want to continue testing the methods of the Mastermind class and it needs us to compare if the SecretCode is correct we have to generate a MockObject so that we can pass the secret code with which we want to do the tests.

The main tests we check are checkSecretCodeTest() in the SecretCodeTest and enterCodeTest() in the MastermindTest, and we don’t need to modify our main class SecretCode to do this.

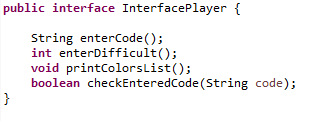


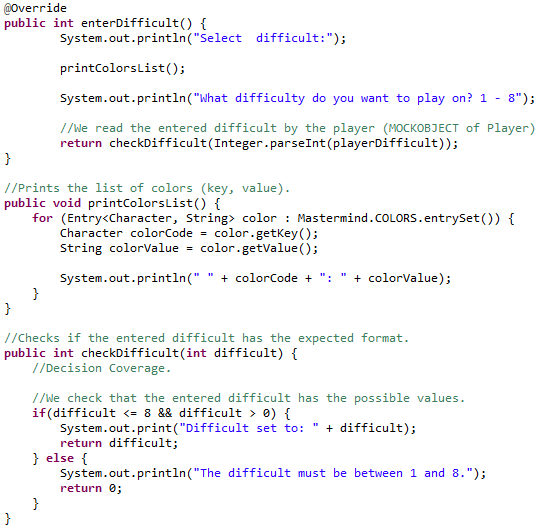
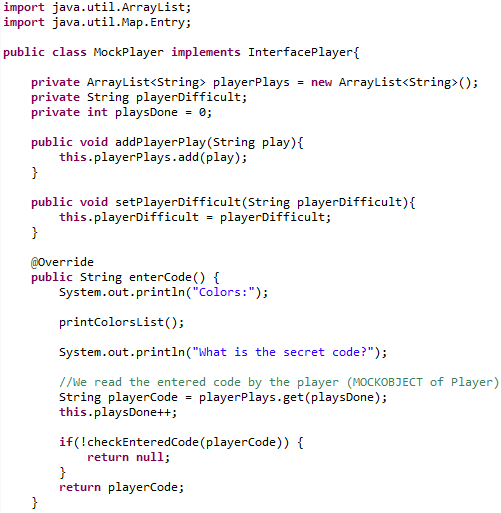


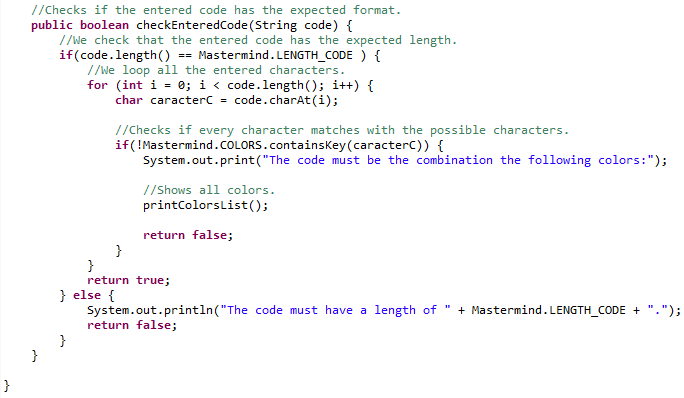
* **MockPlayer**:

We had to implement this MockObject because in the Player class, the enterCode() and the enterDifficult() function required user interaction and we could not automate this process by doing more thorough testing.

The main tests we check are playerSetDifficultTest() and playerPlaysGameTest() both in the MastermindTest.







**Statement Coverage**

Missing statements are due to the functions for which we needed the help of MockObjects. That is because some functions generate random values ​​and others require human interaction and it is not possible to test it without MockObjects. We have also created two classes that run all the tests at the same time (TestRunner and AllTests).

